BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML

Charles Reis[*] *John Dunagan*[†] *Helen J. Wang*[†] *Opher Dubrovsky*[†] *Saher Esmeir*[‡]

Abstract

Vulnerability-driven filtering of network data can offer a fast and easy-to-deploy alternative or intermediary to software patching, as exemplified in Shield [43]. In this paper, we take Shield's vision to a new domain, inspecting and cleansing not just static content, but also dynamic content. The dynamic content we target is the dynamic HTML in web pages, which have become a popular vector for attacks. The key challenge in filtering dynamic HTML is that it is undecidable to statically determine whether an embedded script will exploit the browser at run-time. We avoid this undecidability problem by rewriting web pages and any embedded scripts into safe equivalents, inserting checks so that the filtering is done at run-time. The rewritten pages contain logic for recursively applying run-time checks to dynamically generated or modified web content, based on known vulnerabilities. We have built and evaluated *BrowserShield*, a system that performs this dynamic instrumentation of embedded scripts, and that admits policies for customized run-time actions like vulnerability-driven filtering.

1 Introduction

Web browsers have become an important interface between users and many electronic services such as information access, personal communications, office tasks, and e-commerce. The importance of web browsers is accompanied by rich functionality and extensibility, which arguably have also contributed to their popularity as a vector of attack. During the year 2005, 8 out of 29 critical Microsoft security bulletins, corresponding to 19 vulnerabilities, are due to flaws in Internet Explorer (IE) or its extensions such as ActiveX controls [1]. There were also 6 security bulletins for Firefox [14], corresponding to 59 vulnerabilities over the same period of time.

To date, the primary way to defend browser vulnerabilities is through software patching. However, studies have shown that the deployment of software patches is often delayed after the patches become available. Services such as Windows Update download patches automatically, but typically delay enactment if the patch requires a reboot or application restart. This delay helps both home and corporate users to save work and schedule downtime. An additional delay in the corporate setting is that patches are typically tested prior to deployment, to avoid the potentially high costs for recovering from a faulty patch [5].

As a result, there is a dangerous time window between patch release and patch application during which attackers often reverse-engineer patches to gain vulnerability knowledge and then launch attacks. One study showed that a large majority of existing attacks target known vulnerabilities [4].

For vulnerabilities that are exploitable through application level protocols (e.g., HTTP, RPC), previous work, Shield [43], addresses the patch deployment problem by filtering malicious traffic according to vulnerability signatures at a firewall above the transport layer. The vulnerability signatures consist of a vulnerability state machine that characterizes all possible message sequences that may lead to attacks, along with the message formats that can trigger the exploitation of the application (e.g., an overly long field of a message that triggers a buffer overrun). The key characteristic of this approach is that it cleanses the network data without modifying the code of the vulnerable application. This data-driven approach makes signature deployment (and removal if needed) easier than it is for patches. Vulnerability signature deployment can be automatic rather than user-driven and use the same deployment model as anti-virus signatures.

These desirable features of vulnerability-driven filtering motivated us to explore its potential for exploit removal in web pages. The Shield approach is able to filter static HTML pages by treating HTML as another protocol layer over HTTP. However, the challenge lies in dynamic HTML, where pages can be dynamically generated or modified through scripts embedded in the page — attackers could easily evade Shield filters by using scripts to generate malicious web content at run-time, possibly with additional obfuscation. Determining whether a

[*]University of Washington CS Dept., creis@cs.washington.edu
[†]Microsoft, {jdunagan, helenw, opherd}@microsoft.com
[‡]Technion CS Dept., esaher@cs.technion.ac.il

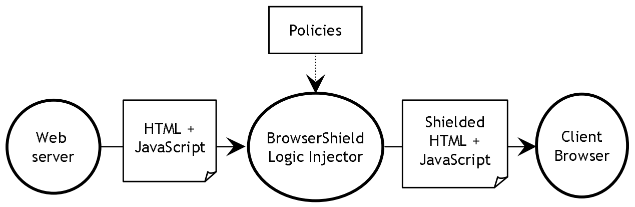


Figure 1: The BrowserShield System

script will eventually exploit a vulnerability is undecidable. Our approach to cleansing dynamic content is to rewrite HTML pages and any embedded scripts into safe equivalents before they are rendered by the browser. The safe equivalent pages contain logic for recursively applying run-time checks to dynamically generated or modified web content, based on known vulnerabilities. To this end, we have designed *BrowserShield*, a system that performs dynamic instrumentation of embedded scripts and that admits policies for changing web page behavior. A vulnerability signature is one such policy, which sanitizes web pages according to a known vulnerability. Figure 1 gives an overview of the BrowserShield system, showing how it transforms HTML and JavaScript using a set of policies. Our system focuses on JavaScript because it is the predominant scripting language used on the web; a full fledged system would require additionally rewriting or disabling VBScript and any other script languages used by web browsers that BrowserShield protects.

Our general approach of code rewriting for interposition has been used in other contexts. Code rewriting has been used to isolate faults of software extensions [41]. Java bytecode rewriting has been used to enable security polices [10, 37], such as stack inspection policies for access control. However, rewriting script code for web browsers poses additional challenges: JavaScript is a prototype-based language, and the combination of this with JavaScript's scoping rules, implicit garbage collection and pervasive reflection required a number of techniques not needed by previous rewriting work in other contexts.

We have designed BrowserShield to adhere to well established principles for protection systems: complete interposition of the underlying resource (i.e., the HTML document tree), tamper-proofness and transparency [3, 10, 33]. In addition, BrowserShield is a general framework that supports applications other than vulnerability-driven filtering. For example, we have authored policies that add UI invariants to prevent certain phishing attempts.

Because BrowserShield protects web browsers by transforming their inputs, not the browser itself, the BrowserShield logic injector can be deployed at client or edge firewalls, browser extensions, or web publishers that republish third-party content such as ads.

We have implemented a prototype of the BrowserShield system, in which the rewriting logic is injected into a web page at an enterprise firewall and executed by the browser at rendering time. Our prototype can transparently render many familiar websites that contain JavaScript (e.g., www.google.com, www.cs.washington.edu, www.mit.edu). We also successfully translated and ran a large intranet portal application (Microsoft SharePoint) that uses 549 KB of JavaScript libraries.

We chose the firewall deployment scenario because it offers the greatest manageability benefit, as BrowserShield updates can be centralized at the firewall, immediately protecting all client machines in the organization without any BrowserShield-related installation at either clients or web servers. The main disadvantage of this deployment scenario is that firewalls have no visibility into end-to-end encrypted traffic. Nevertheless, commercial products [35] already exist that force traffic crossing the organization boundary to use the firewall (instead of a client within the organization) as the encryption endpoint, trading client privacy for aggregate organization security. Also, the browser extension and web publisher deployment scenarios transparently handle encrypted traffic.

Our evaluation focuses on the effectiveness of the BrowserShield design and the performance of our implementation. Our analysis of recent IE vulnerabilities shows that BrowserShield significantly advances the state-of-the-art; existing firewall and anti-virus techniques alone can only provide patch-equivalent protection for 1 of the 8 IE patches from 2005, but combining these two with BrowserShield is sufficient to cover all 8. We evaluated BrowserShield's performance on real-world pages containing over 125 KB of JavaScript. Our evaluation shows a 22% increase in firewall CPU utilization, and client rendering latencies that are comparable to the original page latencies for most pages.

The rest of the paper is organized as follows: In Section 2 we describe a typical browser vulnerability that we would like to filter. We discuss the design of BrowserShield in Section 3, and give BrowserShield's JavaScript rewriting approach in detail in Section 4. We describe our implementation in Section 5. In Section 6 we give our evaluation of BrowserShield. We discuss related work in Section 7, and conclude in Section 8.

2 A Motivating Example

As a motivating example of vulnerability-driven filtering, we consider *MS04-040*: the HTML Elements Vulnerability [28] of IE from December, 2004. In this vulnerability, IE had a vulnerable buffer that was overrun if

```
function (tag) {
  var len = 255; // not the actual limit

  // Look for long attribute values
  if ((contains("name", tag.attrs) &&
       tag.attrs["name"].length > len) &&
      (contains("src", tag.attrs) &&
       tag.attrs["src"].length > len)) {
    // Remove all attributes to be safe
    tag.attrs = [];
    // Return false to indicate exploit
    return false;
  }
  // Return true to indicate safe tag
  return true;
}
```

Figure 2: JavaScript code snippet to identify exploits of the MS04-040 vulnerability

both the `name` and the `src` attributes were too long in an `iframe`, `frame`, or `embed` HTML element.

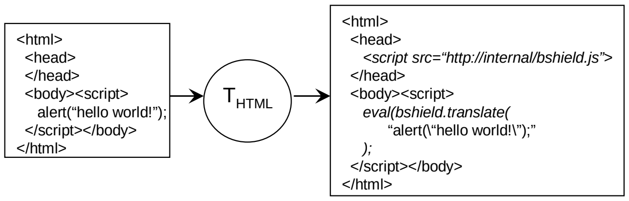
Figure 2 shows a corresponding snippet of JavaScript code that can be used to identify and to remove exploits of this vulnerability. As input, the function takes an object representing an HTML tag, including an associative array of its attributes. When invoked on an `<iframe>`, `<frame>` or `<embed>` tag, the function determines whether the relevant attributes exceed the size of the vulnerable buffer.

The goal of BrowserShield is to take this vulnerability-specific filtering function as a policy and apply it to all occurrences of the vulnerable tags whether they are in static HTML pages or dynamically generated by scripts. The framework could react in many ways to detected exploits; our current system simply stops page rendering and notifies the user. Vulnerability driven filtering, used as a patch alternative or intermediary, should prevent all exploits of the vulnerability (i.e., zero false negatives), and should not disrupt any exploit-free pages (i.e., zero false positives). We design BrowserShield to meet these requirements.

3 Overview

The BrowserShield system consists of a JavaScript library that translates web pages into safe equivalents and a logic injector (such as a firewall) that modifies web pages to use this library.

BrowserShield uses two separate translations along with policies that are enforced at run-time. The first translation, T_{HTML}, translates the HTML: It tokenizes an HTML page, modifies the page according to its policies (such as the one depicted in Figure 2) and wraps the script elements so that the second translation,



```
<html>
  <head>
  </head>
  <body><script>
    alert("hello world!");
  </script></body>
</html>
```

```
<html>
  <head>
    <script src="http://internal/bshield.js">
  </head>
  <body><script>
    eval(bshield.translate(
          "alert(\"hello world!\");"
    );
  </script></body>
</html>
```

Figure 3: T_{HTML} Translation

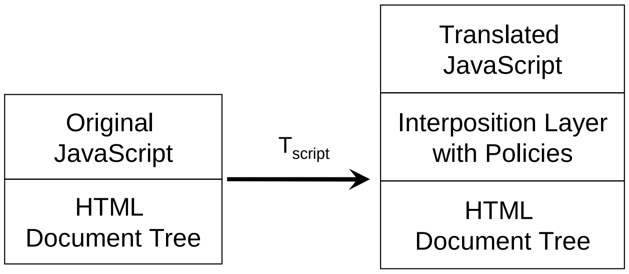


Figure 4: T_{script} Translation

T_{script}, will be applied at run-time during page rendering at the browser. T_{HTML} is depicted in Figure 3 using `bshield.translate(...)` to invoke T_{script}. T_{script}, as depicted in Figure 4, parses and rewrites JavaScript to access the HTML document tree through an interposition layer. This layer regulates all accesses and manipulations of the underlying document tree, recursively applies T_{HTML} to any dynamically generated HTML, and recursively applies T_{script} to any dynamically generated script code. Additionally, the interposition layer enforces policies, such as filtering exploits of known vulnerabilities.

Since users can choose to disable scripting in their web browsers, we must ensure BrowserShield protects such users even without the JavaScript library. We transparently handle such clients by applying T_{HTML} at the logic injector, independent of the user's browser. Any modifications due to T_{script} are still in place, but disabling scripts has made them irrelevant, along with the original script code.

Browser extensions, such as ActiveX controls, can also manipulate the document tree. The security model for such extensions is that they have the same privileges as the browser, and thus we focus on interposing between script and the extensions, not between the extensions and the document tree. This allows BrowserShield to prevent malicious script from exploiting known vulnerabilities in trusted browser extensions.

We have designed BrowserShield to adhere to well established principles for protection systems [3, 10, 33]:

- *Complete interposition*: All script access to the HTML document tree must be mediated by the BrowserShield framework.
- *Tamper-proof*: Web pages must not be able to modify or tamper with the BrowserShield framework in unintended ways.
- *Transparency*: Apart from timing considerations and reasonable increases in resource usage, web pages should not be able to detect any changes in behavior due to the BrowserShield framework. The sole exception is for policy enforcement (e.g., the behavior of a page containing an exploit is visibly modified).
- *Flexible policies*: We desire the BrowserShield framework to have a good separation between mechanism and policy, to make the system flexible for many applications.

4 Design

We now give a detailed discussion of the BrowserShield script library. While much previous work uses code rewriting for interposition [10, 11, 12, 41], our approach is heavily influenced by the fact that our code lives in the same name space as the code it is managing, and also several subtleties of JavaScript. First, JavaScript is a prototype-based language [39], not a class-based language like Java. In prototype-based languages, objects are created using other objects as prototypes, and can then be modified to have a different set of member variables and methods. A consequence of this is that JavaScript has no static typing: different data types can be assigned to the same variable, even for references to functions and object methods. Second, scoping issues must be dealt with carefully, as assigning a method to a new object causes any use of the `this` keyword in the method to bind to the new object. Thus, any interposition mechanisms must ensure that `this` is always evaluated in the intended context. Third, JavaScript uses a garbage collector that is not exposed to the language. Fourth, the language has pervasive reflection features that let a script explore its own code and object properties.

As a result of these subtleties, BrowserShield must use a series of interposition mechanisms: *method wrappers*, *new invocation syntax*, and *name resolution management*. We justify and describe these mechanisms in the following subsections, organized by our goals for the framework.

4.1 Complete Interposition

To provide complete interposition, BrowserShield must mediate all possible accesses and manipulations allowed by the Document Object Model (DOM) over the HTML document trees (including script elements). In this subsection, we detail how we achieve this using script rewriting to interpose on function calls, object method calls, object property accesses, object creation, and control constructs. We summarize our rewriting rules in Table 1.

Function and Object Method Calls There are two ways to rewrite function or method calls for interposition: callee rewriting or caller rewriting.

In callee rewriting, the original function or method definition is first saved under a different name, and then the original function or method is redefined to allow interception before calling the saved original. We call the redefined function the *wrapper*. The benefit of callee rewriting is that the rewritten code is localized — only functions or methods of interest are modified, but not their invocations throughout the code. However, callee rewriting does not work for cases where functions or methods cannot be redefined.

In caller rewriting, the invocation is rewritten to an interposition function without changing the original function's definition. The interposition function looks up the appropriate interposition logic based on the identity of the target function or method. Although caller rewriting causes more pervasive code changes, it can interpose on those functions or methods that cannot be overwritten.

In BrowserShield, we have to use a hybrid of both approaches to accommodate the previously mentioned JavaScript subtleties.

JavaScript contains some native functions that cannot be redefined (e.g., `alert`), which necessitates caller rewriting. The first row of Table 1 shows how BrowserShield indirectly invokes a function with its list of parameter values by passing it to `bshield.invokeFunc(func, paramList)`, where `bshield` is a global object that we introduce to contain BrowserShield library code.

However, using caller rewriting alone for interposing on method calls requires maintaining references to state otherwise eligible for garbage collection. Caller rewriting requires maintaining a map from functions and methods of interest to their associated interposition logic. Maintaining this map as a global table would require maintaining a reference to methods of interest on every object ever created, since each object may correspond to a distinct prototype requiring distinct interposition logic. These global table references would prevent reclamation of objects otherwise eligible for garbage collection, possibly causing pages that render normally without BrowserShield to require unbounded memory. To avoid this, BrowserShield maintains the necessary interposition logic on each method, allowing unused state to be reclaimed.

It might seem tempting to maintain this interposition logic as a property on the object. Unfortunately,

Construct	Original Code	Rewritten Code
Function Calls	`foo(x);`	`bshield.invokeFunc(foo, x);`
Method Calls	`document.write(s);`	`bshield.invokeMeth(document, "write", s);`

Object Properties	`obj.x = obj.y;`	`bshield.propWrite(obj, "x",`
		`  bshield.propRead(obj, "y") );`

Object Creation	`var obj = new MyClass(x);`	`var obj = bshield.createObj(`
		`  "MyClass", [x]);`

`with` Construct	`with (obj) { x = 3; }`
	`// x refers to obj.x`
		`(bshield.undefined(obj.x) ?  x = 3 :`
		`  bshield.propWrite(obj, "x", 3));`

Variable Names	`bshield = x;`	`bshield_ = x;`

`in` Construct	`for (i in obj) {...}`	`for (i in obj) {`
		`  if (i=="bshield") continue; ...`
		`}`

Table 1: Sample Code for BrowserShield Rewrite Rules

aliases to the interposed method can be created, and these aliases provide no reference to the object containing the interposition logic. For example, after "`f = document.write`", any interposition logic associated with `document.write` is not associated with `f`; finding the logic would require a global scan of JavaScript objects. Therefore, we use callee rewriting to install a wrapper for the methods of interest, such as those that insert new HTML. These wrappers are installed by replacing the original method with the wrapper and saving the original method as a property on the wrapper (which is itself an object). Because we interpose on object property accesses, object creation, and method invocations, we can install wrappers when an object is first created or used.

Thus far we have justified caller rewriting for functions and callee rewriting for methods. Because JavaScript allows functions to be aliased as methods on objects (e.g., "`obj.m = eval`"), we also must perform caller rewriting for method calls. The rewritten method invocations can then check for potential aliased functions.

JavaScript scoping introduces additional complexity in method interposition. The original method cannot be simply called from the method wrapper, because saving the original method as a property of the wrapper causes the keyword `this` in the original method to refer to the wrapper rather than the intended object. To avoid this problem, we use a *swapping* technique: The wrapper temporarily restores the original method during the wrapper execution, and then reinstalls the wrapper for the object method before the wrapper returns.

During swapping, the first step is to restore the original method. One challenge here is that the method name may not be the same as when the method wrapper was installed, because methods can be reassigned. We solve this problem again with caller rewriting through the rewritten method invocation syntax `invokeMeth(obj, methName, paramList)`, passing the name of the method to the method wrapper.

The swapping process requires an additional check to handle recursive methods, since otherwise a recursive call would directly invoke the original method rather than the swapped out method wrapper, bypassing any interposition logic on nested calls. To this end, the `invokeMeth` method checks to see if a wrapper is already swapped out. If so, `invokeMeth` invokes the wrapper again, ignoring any swapping logic until the original recursive call completes. Because JavaScript is single threaded, we have not needed to handle concurrency during this process.

Object Properties The HTML document tree can be accessed and modified through JavaScript object property reads and writes. For example, the HTML in a page can be modified by assigning values to `document.body.innerHTML`, and a script element's code can be modified by changing its `text` property. To interpose on such actions, BrowserShield replaces any attempts to read or write object properties with calls to the global `bshield` object's `propRead(obj, propName)` and `propWrite(obj, propName, val)` methods, as shown in Table 1. We use an object's identity at run-time to check whether an assignment will create new HTML or script code. If so, `propWrite` applies either T_{HTML} or T_{script} to the value as needed. These identity checks can be done by calling JavaScript library functions that reveal whether the object is part of the HTML document tree. We ensure that BrowserShield uses the authentic library functions, and not malicious replacements, by creating private aliases of the functions before the script begins to run.

This interposition on property accesses is required for installing wrappers when an object is first accessed. Additionally, while wrappers are swapped out during method execution, `propRead` must ensure that any attempts to access the original method are redirected to the swapped-out wrapper.

Object Creation To ensure that method wrappers are initialized in the case of new object creation, BrowserShield must also rewrite the instantiation of new objects to use the `createObj(className, paramList)` method. The `createObj` method is also responsible for interposing on the JavaScript `Function` constructor, which can create new executable functions from its parameters as follows:

```
f = new Function("x", "return x+1;");
```

In this case, `createObj` applies T_{script} to the code argument before instantiating the function.

Control Constructs For control constructs (e.g., `if-then` blocks, loops, etc.), the bodies of the constructs are translated by T_{script}. The bodies of traditional function constructors (e.g., `function foo() {...}`) are translated by T_{script} as well.

JavaScript's `with` construct presents a special case, as it has the ability to modify scope. As shown in Table 1, free variables within a `with` block are assumed to refer to properties on the designated object, unless such properties are undefined. This construct is purely "syntactic sugar" for JavaScript, and thus we handle this case with a syntactic transformation.

4.2 Tamper-Proof

Preventing scripts from tampering with BrowserShield is challenging because BrowserShield logic lives in the same name space as the code it is managing. To address this, we use *name resolution management* to ensure that all BrowserShield logic is inaccessible.

Variable Names In the common case, variable names in a script can remain unchanged. However, we make the `bshield` name inaccessible to scripts to prevent tampering with the global BrowserShield data structure.

To do this, we rename any variable references to `bshield` by appending an underscore to the end of the name. We also append an underscore to any name that matches the `bshield(_*)` regular expression (i.e., that begins with `bshield` and is optionally followed by any number of underscores). Note that JavaScript places no limit on variable name length.

Reflection Reflection in JavaScript allows script code to explore the properties of objects as well as its own code, using two pervasive language features: the syntax for accessing object properties (such as `myScript.text` or `myScript[i]`), and the JavaScript `in` construct.

In the first case, BrowserShield must hide some object properties, because it maintains per-object interposition state (details given in Section 4.3) on some objects. Such state is stored on a `bshield` property of the object, which we hide using property access interposition. Specifically, if a call to `propRead` or `propWrite` attempts to access a property name beginning with `bshield`, we simply append an underscore to the name, thus returning the property value that the original script would have seen. Since array indices can also be used to access object properties, we must return the appropriate value for the given index.

In the second case, the `in` construct allows iteration through all of an object's properties by name. The `bshield` property of an object must be hidden during the iteration if it is present. Thus, BrowserShield inserts a check as the first line of the iteration loop, jumping to the next item if the property name is `bshield`. This is accomplished using the rewrite rule shown in Table 1.

4.3 Transparency

The BrowserShield framework must also ensure its presence is transparent to the original script's semantics. The techniques for preventing tampering described in Section 4.2 contribute to this goal by making BrowserShield inaccessible. Transparency additionally requires that we present to scripts the context they would have in the absence of BrowserShield.

Shadow Copies Scripts can access both their own script code and HTML, which BrowserShield modifies for interposition. To preserve the intended semantics of such scripts, BrowserShield retains a "shadow copy" of all original code before rewriting it. The shadow copy is stored on a `bshield` property of the object. Interposition on property reads and writes allows the shadow copy to be exposed to scripts for access and modification.

Shadowing translated HTML requires additional care. During T_{HTML} transformation, a policy may rewrite static HTML elements. We must similarly create shadow copies for such translated HTML elements, but we cannot directly create a JavaScript object in HTML to store the shadow copy. Thus, we persist the shadow copy to a `bshield` HTML tag attribute during T_{HTML}, which is later used by the BrowserShield library. For example, a policy function that rewrites link URLs may modify the `href` attribute of `<a>` tags during the T_{HTML} transformation. Then, the persisted shadow copy looks like this:

```
<a href="http://translatedLink"
   bshield="{href:'http://originalLink'}">
```

When BrowserShield looks for the `bshield` property of the DOM object corresponding to this tag, it interprets this string into an actual `bshield` property with a shadow copy for the `href` attribute.

Because scripts can only interact with shadow copies of their code and not modified copies, our transformations are not required to be idempotent. That is, we will never apply T_{HTML} or T_{script} to code that has already been transformed.

Preserving Context The JavaScript `eval` function evaluates a string as script code in the current scope, and any occurrence of the `this` keyword in the string is bound to the current enclosing object. Thus, if `eval` were to be called from within `bshield.invokeFunc`, the `this` keyword might evaluate differently than in the original context.

For this reason, the rewriting rule for functions is actually more complex than shown in Table 1. Instead, the rewritten code first checks if the function being invoked is `eval`. If so, the parameter is translated using T_{script} and then evaluated in the correct context; otherwise, `invokeFunc` is called as described before. Thus, the code is rewritten as follows:

```
bshield.isEval(bshield.func = foo) ?
  eval(bshield.translate(x)) :
  bshield.invokeFunc(bshield.func, x);
```

Note that the function expression `foo` is assigned to a temporary state variable on the `bshield` object, so that the expression is not evaluated a second time in the call to `invokeFunc`.

This check is a special case that is only needed for `eval`, because `eval` is the only native function in JavaScript that accesses `this`. Other native functions, such as `alert` or `parseInt`, do not access `this`, and can be evaluated within `invokeFunc`.

4.4 Flexible policies

The final goal of BrowserShield is to support flexible policy enforcement. This can be achieved by separating mechanism from policy: Our mechanism consists of the rewrite rules for translating HTML and script code, and our policy consists of the run-time checks invoked by the rewritten code. Some run-time checks are critical for complete interposition, such as applying T_{script} to any string passed to `eval` or the `Function` constructor, or applying T_{HTML} to any string passed to `document.write` or assigned to `document.body.innerHTML`. These checks are always applied, regardless of what policy is in place. Because the interposition is policy-driven, our system can be made incrementally complete. For example, if an undocumented API is discovered that can manipulate the document tree, we simply add a new policy to interpose on this API.

The remaining run-time checks are used for enforcing flexible policies, such as the MS04-040 vulnerability filter in Figure 2. Such policy functions are downloaded separately from the remainder of the BrowserShield code, and they can be updated and customized based on the intended application.

Policy functions are given the chance to inspect and modify script behavior at all interposition points, including property reads and writes, function and method invocations, and object creations. We also allow policy writers to introduce new global state and functions as part of the global `bshield` object, or introduce local state and methods for all objects or for specific objects. Policy functions for HTML can also be registered by tag name. The tags are presented to HTML policy functions as part of a token stream of tags and text, without a full parse tree. It is also possible for policy functions to further parse the HTML token stream to gain additional context, although we have not yet encountered a need for this in the policies we have authored.

5 Implementation

We have implemented a prototype of BrowserShield as a service deployed at a firewall and proxy cache. Our prototype consists of a standard plugin to Microsoft's Internet Security and Acceleration (ISA) Server 2004 [21], and a JavaScript library that is sent to the client with transformed web documents. The ISA plugin plays the role of the BrowserShield logic injector.

We implemented our ISA plugin in C++ with 2,679 lines of code. Our JavaScript library has 3,493 lines (including comments). Most of the ISA plugin code is devoted to parsing HTML, while about half of the JavaScript library is devoted to parsing HTML or JavaScript. This is a significantly smaller amount of code than in a modern web browser, which implies that our trusted computing base is small compared to the code base we are protecting.

The ISA plugin is responsible for applying the T_{HTML} transformation to static HTML. The ISA plugin first inserts a reference to the BrowserShield JavaScript library into the web document. Because this library is distributed in a separate file, clients automatically cache it, reducing network traffic on later requests. T_{HTML} then rewrites all script elements such that they will be transformed using T_{script} at the client before they are executed. Figure 3 depicts this transformation; note that it does not require translating the JavaScript at the firewall.

In our implementation, the firewall component applies T_{HTML} using a streaming model, such that the ISA Server can begin sending transformed data to the client before the entire page is received. This streaming model also means that we do not expect the filter to be vulnerable to state-holding DoS attacks by malicious web pages.

One complexity is that BrowserShield's HTML parsing and JavaScript parsing must be consistent with that of the underlying browser. Any inconsistency will cause false positives and false negatives in BrowserShield runtime checks. For our prototype, we have sought to match IE's behavior through testing and refinement. If future versions of browsers exposed this logic to other programs, it would make this problem trivial.

When the browser starts to run the script in the page, the library applies T_{script} to each piece of script code, translating it to call into the BrowserShield interposition layer. This may sometimes require decoding scripts, a procedure that is implemented in publicly available libraries [34] and which does not require cryptanalysis, though we have not yet incorporated it in our implementation.

A final issue in T_{script} is translating scripts that originate in source files linked to from a source tag. T_{HTML} rewrites such source URLs so that they are fetched through a proxy. The proxy wraps the scripts in the same way that script code embedded directly in the page is wrapped. For example, a script source URL of `http://foo.com/script.js` would be translated to `http://rewritingProxy/translateJS.pl?url=http://foo.com/script.js`. T_{script} is then applied at the client after the script source file is downloaded.

6 Evaluation

Our evaluation focuses on measuring BrowserShield's vulnerability coverage, the complexity of authoring vulnerability filters, the overhead of applying the BrowserShield transformations at firewalls, and the overhead of running the BrowserShield interposition layer and vulnerability filters at end hosts.

6.1 Vulnerability Coverage

We evaluated BrowserShield's ability to protect IE against all critical vulnerabilities for which Microsoft released patches in 2005 [1]. Of the 29 critical patches that year, 8 are for IE, corresponding to 19 IE vulnerabilities. These vulnerabilities fall into three classes: IE's handling of (i) HTML, script, or ActiveX components, (ii) HTTP, and (iii) images or other files. Table 2 shows how many vulnerabilities there were in each area, and whether BrowserShield or another technology could provide patch-equivalent protection. The BrowserShield design is focused on HTML, script, and ActiveX controls, and it can successfully handle all 12 of these vulnerabilities. This includes vulnerabilities where the underlying programmer error is at a higher layer of abstraction than a buffer overrun, e.g., a cross-domain scripting vulnerability. Handling HTTP accounted for 3 of the 19 vulnerabilities. Perhaps surprisingly, 2 out of 3 of these vulnerabilities required BrowserShield in addition to an existing HTTP filter, such as Snort [38] or Shield [43]. This is because malformed URLs could trigger the HTTP layer vulnerabilities regardless of whether the URL came over the network or was generated internally by the browser. BrowserShield is able to prevent the HTML/script layer from triggering the generation of these bad HTTP requests. Processing images or other files accounted for the remaining 4 vulnerabilities. Patch-equivalent protection for these vulnerabilities is already available using existing anti-virus solutions [13].

vulnerability		**protected by**		
type	**#**	**BrowserShield**	**HTTP filter**	**antivirus**
HTML, script, ActiveX	12	12	0	0
HTTP	3	2*	3*	0
images and other files	4	0	0	4

Table 2: BrowserShield Vulnerability Coverage. *Two of the HTTP vulnerabilities required both BrowserShield and an HTTP filter to provide patch-equivalent protection.

Because management and deployment costs are often incurred on a per-patch basis, we also analyze the vulnerabilities in Table 2 in terms of the corresponding patches. For the 8 IE patches released in 2005, combining BrowserShield with standard anti-virus and HTTP filtering would have provided patch-equivalent protection in every case, greatly reducing the costs associated with multiple patch deployments. In the absence of BrowserShield, anti-virus and HTTP filtering would have provided patch-equivalent protection for only 1 of the IE patches.

6.2 Authoring Vulnerability Filters

To evaluate the complexity of vulnerability filtering, we choose three vulnerabilities from three different classes: HTML Elements Vulnerability (MS04-040), COM Object Memory Corruption (MS05-037), and Mismatched DOM Object Memory Corruption (MS05-054).

We filtered for the MS04-040 vulnerability using the function shown in Figure 2. Registering this filter for each of the three vulnerable tags is as simple as:

```
bshield.addHTMLTagPolicy("IFRAME", func);
```

COM object vulnerabilities typically result from IE instantiating COM objects that have memory errors in their constructors. The IE patch blacklists particular COM objects (identified by their `clsid`). Implementing an equivalent blacklist requires adding checks for an HTML tag (the `OBJECT` tag) and sometimes a JavaScript function (the `ActiveXObject` constructor, which can be used to instantiate a subset of the COM objects accessible through the `OBJECT` tag). In the case of MS05-037, it does not appear to be possible to instantiate the vulnerable COM object using the `ActiveXObject` constructor. The `OBJECT` tag filter is conceptually similar to the function shown in Figure 2.

The MS05-054 vulnerability results when the `window` object, which is not a function, is called as a function in the outermost scope. Our interposition layer itself prevents `window` from being called as a function in the outermost scope since all function calls are mediated by BrowserShield with `invokeFunc`. Hence there is no need for a filter. Nevertheless, if this vulnerability had not depended on such a scoping constraint, we could simply have added a filter to prevent calling the object as a function.

To test the correctness of our vulnerability filters, we installed an unpatched image of Windows XP Pro within a virtual machine, and created web pages for each of the vulnerabilities that caused IE to crash. Applying BrowserShield with the filters caused IE to not crash upon viewing the malicious web pages. We tested the fidelity of our filters using the same set of URLs that we used in our evaluation of BrowserShield's overhead (details are in Section 6.3). Under side-by-side visual comparisons, we found that the filters had not changed the behavior of any of the web pages, as desired.

6.3 Firewall Performance

We evaluated BrowserShield's performance by scripting multiple IE clients to download web pages (and all their embedded objects) through an ISA server running the BrowserShield firewall plugin. The ISA firewall ran on a Compaq Evo PC containing a 1.7GHz Pentium 4 microprocessor and 1 GB RAM. Because we are within a corporate intranet, our ISA server connected to another HTTP proxy, not directly to web sites over the internet. We disabled caching at our ISA proxy, and we fixed our IE client cache to contain only the BrowserShield JavaScript library, consistent with the scenario of a firewall translating all web sites to contain a reference to this library.

We ran 10 IE processes concurrently using 10 pages that IE could render quickly (so as to increase the load on the firewall), and repeatedly initiated each page visit every 5 seconds. We used manual observation to determine when the load on the ISA server had reached a steady state.

We chose these 10 pages out of a set of 70 URLs that are the basis for our client performance macrobenchmarks. This set is based on a sample of 250 of the top 1 million URLs clicked on after being returned as MSN Search results in Spring 2005, weighted by clickthrough count. Specifically, the 70 URLs are those that BrowserShield can currently render correctly; the remaining URLs in the sample encountered problems due to incomplete aspects of our implementation, such as JavaScript parsing bugs.

resource	unmodified	browsershield
cpu utilization	15.0%	18.3%
virtual memory	317 MB	319 MB
working set	45.5 MB	46.6 MB
private bytes	26.3 MB	27.3 MB

Table 3: BrowserShield Firewall overheads. "Virtual memory" measures the total virtual memory allocated to the process; "working set" measures memory pages that are referenced regularly; "private bytes" measures memory pages that are not sharable.

We measured CPU and memory usage at the firewall, as shown in Table 3. CPU usage increased by about 22%, resulting a potential degradation of throughput by 18.1%; all aspects of memory usage we measured increased by negligible amounts. We also found that network usage increased only slightly (more detail in Section 6.4.2).

6.4 Client Performance

We evaluated the client component of our BrowserShield implementation through microbenchmarks on the JavaScript interposition layer and macrobenchmarks on network load, client memory usage, and the latency of page rendering.

6.4.1 Microbenchmarks

We designed microbenchmarks to measure the overhead of individual JavaScript operations after translation. Table 4 lists our microbenchmarks and their respective BrowserShield slow-down. Our results are averages over 10 trials, where each trial evaluated its microbenchmark repeatedly, and lasted over half a second. For the first 11 micro-benchmarks, the standard deviation over the 10 trials was less than 2%. In the last case it was less than 8%. The slowdown ratio was computed using the average time required per microbenchmark evaluation with and without the interposition framework.

	operation	slowdown
1	i++	1.00
2	a = b + c	1.00
3	if	1.07
4	string concat ('+')	1.00
5	string concat ('concat')	61.9
6	string split ('split')	21.9
7	no-op function call	44.8
8	x.a = b (property write)	342
9	eval of minimal syntactic structure	47.3
10	eval of moderate syntactic structure, minimal computation	136
11	eval of moderate syntactic structure, significant computation	1.34
12	image swap	1.07

Table 4: BrowserShield Microbenchmarks. Slowdown is the ratio of the execution time of BrowserShield translated code and that of the original code.

Microbenchmarks 1-4 measure operations for which we expect no changes during rewriting, and hence no slowdown. The only slowdown we measure is in the case of the `if` statement. Further examination showed that the BrowserShield translation inserted a semi-colon (e.g., `var a = 1 (linebreak)` changed to `var a = 1; (linebreak)`). This results in a 7% slowdown.

Microbenchmarks 5-8 measure operations we expect to incur a slowdown comparable to an interpreter's slowdown. As detailed in Section 4, BrowserShield translation introduces additional logic around method calls, function calls, and property writes, leading to a slowdown in the range of 20x-400x. This slowdown is in line with good interpreters [32], but worse than what is achieved by rewriting systems targeting other languages, e.g., Java bytecode [10]. BrowserShield is paying a price for the JavaScript subtleties that previous rewriting systems did not have to deal with. We were curious about the difference in slowdown between the two string methods; an additional experiment showed that the difference can be attributed to the JavaScript built-in `concat` method requiring about 3 times as much CPU as the built-in `split` method. Also, it is not surprising that property writes have a greater slowdown than function or method calls because property writes need to both guard the BrowserShield namespace and interpose on writes to DOM elements (such as the text property of scripts).

Microbenchmarks 9-11 explore the overhead of translating JavaScript code of various complexity. The "eval of minimal syntactic structure" microbenchmark measures the cost of translating and then evaluating a simple assignment. The cause of the large slowdown is the additional work done by `eval` in the BrowserShield framework: parsing, constructing an AST, modifying the AST, and outputting the new AST as a JavaScript program. The two subsequent "eval of moderate syntactic structure" microbenchmarks measure the cost of translating and evaluating a simple `for(;;)` loop. This simply demonstrates that as the cost of the computation inside the simple loop increases, the cost of translating the code can decrease to a small fraction of the overall computational cost.

The last microbenchmark measures the overhead of performing a simple manipulation of the DOM – swapping two 35 KB images. This microbenchmark is designed to measure the relative importance of overheads in the JavaScript engine when the JavaScript is manipulating the layout of the HTML page. The JavaScript code to swap these two images requires two property writes (i.e., `img.src = 'newLink'`), and we described above how BrowserShield translation adds significant overhead to property writes. Nonetheless, the overall slowdown is less than 8%. In particular, the raw time to swap the image only increases from 26.7 milliseconds to 28.5 milliseconds. This suggests that even the large overheads that BrowserShield translation adds to some language constructs may still be quite small in the context of a complete web page.

In summary, BrowserShield incurs a significant overhead on the language constructs where it must add interpreter-like logic, but these overheads can be quite small within the context of the larger DOM manipulations in embedded scripts.

6.4.2 Macrobenchmarks

We designed macrobenchmarks to measure the overall client experience when the BrowserShield framework is in place. In particular, the macrobenchmarks include all the dynamic parsing and translation that occurs before the page is rendered, while the microbenchmarks primarily evaluated the performance of the translated code accomplishing a task relative to the untranslated code accomplishing that same task. To this end, we scripted an instance of IE to download each of the 70 web pages in our workload 10 times. For the same reasons given in our evaluation of the BrowserShield ISA component, we maintained that the only object in the IE cache was the BrowserShield JavaScript library. These caching policies represent a worst-case for client latency. This measurement includes the overhead of the three filters that we discussed in Section 6.1. We then repeated these measurements without the BrowserShield framework and translation.

We set a 30 second upper limit on the time to render the web page, including launching secondary (popup) windows and displaying embedded objects, but not waiting for secondary windows to render. We visually verified that the programmatic signal that rendering had completed indeed corresponded to the user's perception that the page had rendered. IE hit the 30-second timeout several times in these trials, and it hit the timeouts both when the BrowserShield framework and translation were present and when the framework and translation were absent. We did not discern any pattern in these timeouts, and because our experiments include factors outside our control, such as the wide-area network and the servers originating the content, we do not expect page download times to be constant over our trials. We re-ran the trials that experienced the timeouts.

Figure 5 shows the CDF of page rendering with and without BrowserShield. On average BrowserShield added 1.7 seconds to page rendering time. By way of contrast, the standard deviation in rendering time without BrowserShield was 1.0 seconds.

In Figure 6, we further break down the latency for the 10 pages that took the most time to render under BrowserShield. They experienced an average increase in latency of 6.3 seconds, requiring 3.9 seconds on average without BrowserShield and 10.2 seconds on average with

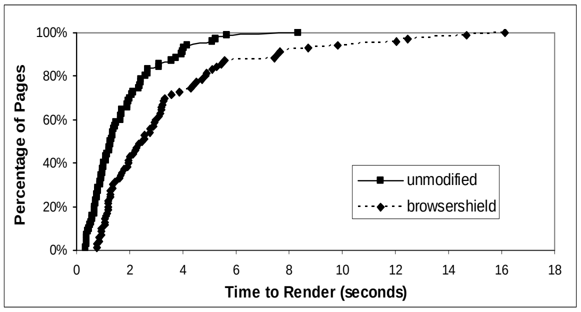


Figure 5: Latency CDF with and without BrowserShield

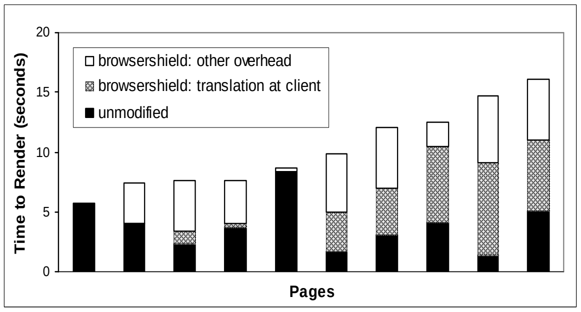


Figure 6: Breakdown of latency for slowest 10 pages under BrowserShield

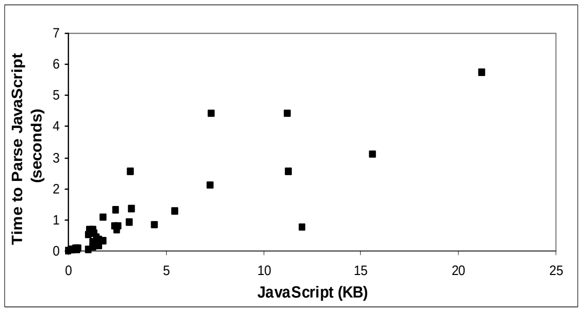


Figure 7: Latency of JavaScript parsing

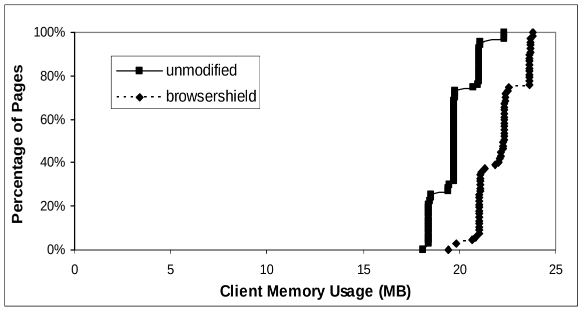


Figure 8: Memory Usage at Client

BrowserShield. Of this 6.3 seconds of increased latency, we found that 2.8 seconds (45%) could be attributed to the overhead of dynamically translating JavaScript and HTML within IE. We attribute the remaining overhead to effects such as the overhead of evaluating the translated code, and the time to modify the HTML at the firewall.

We broke down the latency of dynamic translation for both HTML and JavaScript into 2 parts each: time to parse the JavaScript/HTML into an AST and convert the modified AST back to a string, and the time to modify the AST. We found that the time to parse the JavaScript to and from a string was always more than 70% of the overall latency of dynamic translation, and it averaged 80% of the overall latency. Figure 7 shows the JavaScript parsing time versus the number of kilobytes. Fitting a least-squares line to this data yields an average parse rate of 4.1 KB of JavaScript per second, but there was significant variation; the slowest parse rate we observed was 1.3 KB/second.

Figure 8 shows the memory usage of page rendering with and without BrowserShield. We found that private bytes (memory pages that are not sharable) was the client memory metric that increased the most when rendering the transformed page. Private memory usage increased on average by 11.8%, from 19.8 MB to 22.1 MB. This increase was quite consistent; no page caused memory usage to increase by more than 3 MB.

We also measured the increased network load over a single run through the pages both with and without BrowserShield. We measured an average increase of 9 KB, less than the standard deviation in the network load over any individual trial due to background traffic during our measurements. We expect BrowserShield rewriting to only slightly increase the network load, because the firewall just adds script wrappers, while the translation itself happens at the client.

7 Related Work

We first compare with other protection systems in Section 7.1. We then discuss BrowserShield's relation to the extensive work on code rewriting and interposition in Section 7.2.

7.1 Remote Exploit Defense

In our prior work on Shield [43], we proposed using vulnerability-specific filters to identify and block network traffic that would exploit known software vulnerabilities. Shield maintains protocol-specific state machines in an end-host's network stack, allowing it to recognize when a packet will trigger a vulnerability. However, the Shield approach does not address dynamic content such as scripts in web documents, since it is undecidable whether script code in a document will eventually exploit a vulnerability. BrowserShield shares Shield's focus on vulnerability-specific filters, but in contrast to

Shield, its use of runtime interposition allows it to handle exploits in dynamic HTML.

Like BrowserShield and Shield, IntroVirt also employs vulnerability-specific predicates, specifically to detect past and present intrusions using virtual machine introspection and replay [23]. As a result, IntroVirt allows "just in time" patch application: postponing the application of a patch while an exploit has not occurred, and rolling back system state to apply a patch if an exploit does occur. BrowserShield instead offers protection while a patch is being tested (or otherwise delayed) by the administrator of a vulnerable system, buying time even in cases where exploits are immediately attempted. Additionally, BrowserShield supports more flexible deployment scenarios. For example, it does not require the client's browser to run inside an instrumented virtual machine.

Opus [2] seeks to address the problem of patching by allowing patches to be applied without restarting the application. Opus provides tools for developers to increase the reliability of such "dynamic" patches during development. However, these tools reduce, but do not eliminate, the programmer's burden to produce a correct dynamic patch.

Vigilante [7] focuses on worm containment, automatically detecting and preventing the spread of worm traffic. Vigilante combines rapid distribution of self-certifying alerts and automatic filter generation, along with vulnerability detection techniques such as non-executable pages [30] and dynamic dataflow analysis [29]. These techniques, even with the Vigilante improvements, admit false negatives. BrowserShield does not share the speed constraint of Vigilante, since browser exploits require human involvement and therefore do not spread on the same time scales as worms. Therefore, we are able to trade off the speed of automatically generated vulnerability filters for the accuracy of hand-coded filters.

EarlyBird [36] and Autograph [24] are two exemplary systems that use pattern matching to block network traffic containing exploits. Pattern matching scales to high data rates, crucial to the authors' goal of stopping worm outbreaks at network choke points. The HTML scripts that are BrowserShield's focus seem difficult to detect consistently with pattern matching, as they can trivially modify themselves at the client.

HoneyMonkey [44] aims to discover web servers that distribute malicious code. In HoneyMonkey, virtual machines automatically fetch content from the web and use black-box techniques to discover exploits. Exploit discovery is complimentary to BrowserShield's approach of providing patch-equivalent protection to clients.

Finally, a number of techniques have aimed to sandbox the browser or other applications, in effect protecting the operating system from the impact of an exploit. These techniques include system call interposition [17, 18, 22] and Microsoft's "protected mode" for IE in Windows Vista [20]. These may limit damage to a user's computing environment, but they do not protect the browser itself, allowing attacks such as keylogging to easily be conducted from exploited browsers. Tahoma [8] takes the confinement approach one step further, sandboxing browsers in virtual machines and using site-specific manifests to restrict browser traffic to known servers. While this could help to mitigate many browser related problems, the difficulty of getting such manifests widely adopted is unclear.

7.2 Interposition Techniques

Interposition techniques such as code rewriting have been used in previous work to achieve additional safety properties or otherwise modify the behavior of existing code. Code rewriting is only one of several alternatives for backward compatible modifications, and the choice of technique is influenced by tradeoffs in deployability and performance. Directly modifying the execution environment, such as the Java Virtual Machine, has the highest deployment barriers. Some work instead uses a level of indirection, such as emulation (e.g, Bochs [6]), easing deployment but incurring a high performance overhead. Thus, BrowserShield and others [10, 41, 42] employ code rewriting, with its low barriers to deployment and smaller performance overhead than that required by an emulator.

We characterize interposition techniques by the target of interposition, since the technical differences between targets require different solutions. Compared to approaches for other interposition targets, BrowserShield must address a new combination of technical challenges presented by JavaScript: its scoping rules, an implicit garbage collector, pervasive reflection, and its prototype-based object model (which implies a lack of static typing).

Machine Code Many approaches focus on the machine code interface, whether rewriting binary instructions or emulating them at runtime. Software Fault Isolation (SFI) [41] rewrites binary code to insert runtime checks, creating sandboxes that prevent code from writing or jumping to addresses outside its fault domain. This creates process-like memory boundaries between units of code within a process. The more recent XFI [9] uses binary rewriting to provide flexible access control and additional integrity guarantees. VMware ESX Server [42] also rewrites machine code, in its case to allow programs to be virtualized on x86 hardware. Etch [31] rewrites machine code with the goals of profiling and measurement. Valgrind [40] and Program Shepherding [25] are dynamic binary instrumentation tools. Valgrind's goal is to offer debugging and profiling support, while Program Shepherding's goal is to monitor control flow, preventing the transfer of control to data regions which might include malicious code.

The techniques used for rewriting at the machine code interface do not need to address any of the four challenges of JavaScript rewriting that have influenced BrowserShield: scoping, reflection, garbage collection or typing. Most work interposing at the machine code interface only adds semantics that can be defined in terms of low level operations, such as enforcing a process-like memory boundary, as in SFI. Indeed, Erlingsson and Schneider [11] note the difficulty of extending rewriting at the machine code interface to enforce policies on the abstractions internal to an application. BrowserShield's interposition target (the HTML document tree) is such an application-internal abstraction.

System Call Interface Much previous work has modified user level program behavior by interposing on the system call interface. Jones introduces a toolkit for system call interposition agents that simplifies tasks such as tracing, emulation, and sandboxing [22]. Wagner et al. use system call interposition in Janus to confine untrusted applications to a secure sandbox environment [18]. Garfinkel notes difficulties in trying to interpose on the system call interface [16], such as violating OS semantics, side effects, and overlooking indirect paths. Garfinkel et al. discuss a delegation-based architecture to address some of these problems [17]. Naccio describes an approach to provide similar guarantees by rewriting x86 code that links against the Win32 system call interface [12]. Naccio can also rewrite Java bytecode.

Work on the system call interface differs from BrowserShield both in goal and in technique. System call interposition can guard external resources from an application, while the goal of BrowserShield is to guard an application-internal resource, the HTML document tree. Naccio's use of rewriting as a technique to interpose on the system call interface does not present any of the four technical challenges (scoping, reflection, garbage collection or typing) relevant to JavaScript rewriting. For example, Naccio also wraps methods to accomplish interposition, but Naccio's method wrappers do not need to handle JavaScript's scoping rules, and so do not need to implement swapping.

Java Bytecode Several pieces of previous work [10, 11, 37], including the previously mentioned Naccio [12], have used rewriting at the Java Virtual Machine bytecode interface [26]. This interface is type-safe, and provides good support for reasoning about application-internal abstractions. In the most similar of these works to BrowserShield, Erlingsson's PoET mechanism rewrites Java bytecode to enforce security policies expressed in the PSLang language [10].

JavaScript's pervasive reflection, scoping rules, and prototype-based object model forced us to develop several techniques not needed for Java bytecode rewriting. For example, where Java bytecode rewriting can interpose on Java's reflection API, BrowserShield must interpose on all property reads and writes, as well as some `for` loops, to achieve similar control over reflection. Additionally, Java bytecode rewriting can achieve complete interposition by only modifying callees (using method wrappers) and without maintaining state, though some previous work allowed modifying callers or adding state to simplify policy construction [10]. In contrast, BrowserShield must modify both callers and callees to appropriately handle scoping and the possibility of functions aliased as methods (and vice versa). Also, BrowserShield must maintain state, requiring careful attention to its interaction with the JavaScript garbage collector.

Web Scripting Languages We are not aware of any full interposition techniques for web scripting languages like JavaScript. The SafeWeb anonymity service used a JavaScript rewriting engine that failed to provide either complete interposition or transparency [27]. The Greasemonkey [19] extension to the Firefox browser allows users to run additional site-specific scripts when a document is loaded, but it does not provide complete interposition between existing script code and the HTML document tree.

8 Conclusion

Web browser vulnerabilities have become a popular vector of attacks. Filtering exploits of these vulnerabilities is made challenging by the dynamic nature of web content. We have presented BrowserShield, a general framework that rewrites HTML pages and any embedded scripts to enforce policies on run-time behavior. We have designed BrowserShield to provide complete interposition over the underlying resource (the HTML document tree) and to be transparent and tamper-proof. Because BrowserShield transforms content rather than browsers, it supports deployment at clients, firewalls, or web publishers. Our evaluation shows that adding this approach to existing firewall and anti-virus techniques increases the fraction of IE patches from 2005 that can be protected at the network level from 12.5% to 100%, and that this protection can be done with only moderate overhead.

We have focused on the application of vulnerability-driven filtering in this paper, but JavaScript rewriting techniques may also enable new functionality for AJAX (Asynchronous JavaScript and XML) applications. Some potential uses include: eliminating the effort currently required to modify a website for the Coral [15]

CDN; modifying the cached search results returned by web search engines to redirect links back into the cache (since the original site may be unavailable); allowing appropriately sandboxed dynamic third-party content on a community site (such as a blog or wiki) that currently must restrict third-party content to be static; and debugging JavaScript code when attaching a debugger is infeasible, perhaps offering call traces or breakpoint functionality for complex scripts. User interface changes could even be added to make phishing more difficult, e.g., enforcing the display of origin URLs on all pop-up windows. As this list suggests, we are optimistic that JavaScript rewriting is a widely applicable technique.

References

[1] Microsoft Security Bulletin Summaries and Webcasts, 2005. http://www.microsoft.com/technet/security/bulletin/summary.mspx.

[2] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online Patches and Updates for Security. In *Usenix Security*, 2005.

[3] J. P. Anderson. Computer Security Technology Planning Study Volume II. ESD-TR-73-51, Vol. II, Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford, MA, October 1972.

[4] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of Vulnerability: a Case Study Analysis. *IEEE Computer*, December 2000.

[5] S. Beattie, S. Arnold, C. Cowan, P. Wagle, and C. Wright. Timing the Application of Security Patches for Optimal Uptime. In *LISA*, 2002.

[6] Bochs. http://bochs.sourceforge.net/.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *SOSP*, 2004.

[8] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A Safety-Oriented Platform for Web Applications. In *IEEE Symposium on Security and Privacy*, 2006.

[9] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, 2006.

[10] Ú. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, 2000.

[11] Ú. Erlingsson and F. B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *WNSP: New Security Paradigms Workshop*, 2000.

[12] D. Evans and A. Twyman. Flexible Policy-Directed Code Safety. In *IEEE Symposium on Security and Privacy*, 1999.

[13] Eweek: Anti-Virus Protection for WMF Flaw, December 2005. http://www.eweek.com/article2/0,1895,1907102,00.asp.

[14] Mozilla Security Alerts and Announcements. http://www.mozilla.org/security/.

[15] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing Content Publication with Coral. In *NSDI*, 2004.

[16] T. Garfinkel. Traps and Pitfalls: Practical Problems in in System Call Interposition based Security Tools. In *NDSS*, 2003.

[17] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*, 2004.

[18] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Usenix Security*, 1996.

[19] Greasemonkey. http://greasemonkey.mozdev.org/.

[20] Protected Mode in Vista IE7. http://blogs.msdn.com/ie/archive/2006/02/09/528963.aspx.

[21] ISA Server. http://www.microsoft.com/isaserver/default.mspx.

[22] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *SOSP*, 1993.

[23] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions Through Vulnerability-specific Predicates. In *SOSP*, 2005.

[24] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Usenix Security*, 2004.

[25] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution via Program Shepherding. In *Usenix Security*, 2002.

[26] T. Lindholm and F. Yellin. The Java Virtual Machine Specification, 2nd edition, 1999.

[27] D. Martin and A. Schulman. Deanonymizing Users of the SafeWeb Anonymizing Service. In *USENIX Security*, 2002.

[28] Microsoft Security Bulletin MS04-040, December 2004. http://www.microsoft.com/technet/security/Bulletin/MS04-040.mspx.

[29] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.

[30] Pax. http://pax.grsecurity.net/.

[31] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Usenix NT Workshop*, 1997.

[32] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. The Structure and Performance of Interpreters. In *ASPLOS*, 1996.

[33] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *SOSP*, 1973.

[34] Windows Script Decoder. http://www.virtualconspiracy.com.

[35] Secure Computing. http://www.securecomputing.com/pdf/WW-SSLscan-PO.pdf.

[36] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *OSDI*, 2004.

[37] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *SOSP*, 1999.

[38] The Open Source Network Intrusion Detection System. http://www.snort.org/.

[39] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *OOPSLA*, 1987.

[40] Valgrind. http://www.valgrind.org/.

[41] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *SOSP*, 1993.

[42] C. A. Waldspurger. Memory Resource Management in VMware ESX Server . In *OSDI*, 2002.

[43] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *SIGCOMM*, 2004.

[44] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *NDSS*, 2006.